Module 02: Key Exchange and the TLS 1.3 Protocol

Week-05: Implementing the TLS 1.3 Protocol Handshake and Session Resumption

Jan Gilcher, Fernando Virdia, Kenny Paterson, and Lukas Burkhalter

jan.gilcher@inf.ethz.ch, fernando.virdia@inf.ethz.ch
kenny.paterson@inf.ethz.ch, lubu@inf.ethz.ch

October 2021

Welcome to the second lab for this module. This lab is part of the Module-02 on *Key Exchange and TLS 1.3*. Due to technical limitations the labs will only take place in person. You are also welcome to post your questions and to join ongoing discussions on the Moodle forum at the following link:

`https://moodle-app2.let.ethz.ch/mod/forum/view.php?id=616927`

Students who cannot attend the labs in person are especially encouraged to use Moodle to ask questions.

Overview

This lab serves as an opportunity to investigate and implement a (streamlined) version of one of the most important cryptographic protocols in use today – the Transport Layer Security Protocol, version 1.3 (throughout the lab we will refer to this simply as TLS 1.3). In this lab, we will use the symmetric and asymmetric cryptographic primitives you have seen in previous weeks, and use them to implement various cryptographic primitives specific to TLS 1.3. We will also be helping to implement client functions for the Handshake Protocol – an "authenticated key exchange" (AKE) protocol.

As described by the TLS 1.3 RFC [1], the handshake protocol "*authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.*"

We have split this lab into three parts. The first part we already did last week: implementing the cryptographic functions needed in TLS 1.3. In the second part, we will be implementing client functions for the Handshake Protocol to enable client and server to perform a single handshake and then send encrypted messages to each other. In the third part, we will extend both the client and server implementation of the Handshake Protocol to support session resumption and sending of 0-RTT data.

Please note that we expect you to use Python-3 (>=3.4) for this lab. We will not accept submissions/solutions coded in any other language, including older versions of Python. *So please make sure that your submissions are Python-3 compatible.*

The focus of this lab is on cryptographic APIs, as well as engaging with formal specification documents and cryptographic documentation. As such we expect you to read the relevant sections of the TLS 1.3 RFC, which will usually be referenced. We hope that this lab will give you insight into the experience of coding real-world applications and applied cryptography.

Getting Started

In case you haven't already done this last week, for this lab you will need to install an elliptic-curve library using `pip3 install tinyec`, and modern AEAD libraries using `pip3 install pycryptodomex`. Note that if you encounter errors during the installation, it may be worth trying to `pip3 uninstall pycrypto`. Similarly to the last module, we are also providing a virtual machine (VM) image that has the necessary dependencies pre-installed and ready to use.

The link to access and download the VM is available on Moodle. Below we provide you with a simple set of instructions on how to use this VM:

- Boot the VM using your favourite virtualization software. For example, you can download and use the Oracle VM Virtualbox from the url below:

  `https://www.virtualbox.org/`

- Login using the root password `isl`
- At this point, you can run any Python script(s) that are required for this assessment.

Note that when automatically evaluating your code, we will use the same build environment. So using the VM would additionally allow you to pre-test your submission in an environment resembling our automated testing environment.

Before we begin, we recap the cryptographic background and notation that we will be using throughout this lab sheet.

Background and Notations

- $(x, X = g^x) \leftarrow_R \mathsf{DH.KeyGen}(\lambda)$: denotes the Diffie-Hellman key generation algorithm. It takes as input a security parameter λ, and outputs a secret/public Diffie-Hellman pair $(x, X = g^x)$.
- $(g^{xy}) \leftarrow \mathsf{DH.DH}(x, Y)$: denotes the Diffie-Hellman computation algorithm. It takes as input a Diffie-Hellman secret value x and a Diffie-Hellman public value Y, and outputs a shared secret g^{xy}.

  For all implementations in this lab, we will be using elliptic-curve Diffie-Hellman to perform DH operations, imported from **tinyec**. For greater readability and ease of exposition, we will be using the exponential notation for Diffie-Hellman throughout i.e. we write g^x in place of scalar multiplication $[x]P$ where P is a base point on an elliptic curve.

- $Y \leftarrow \mathsf{H}(X)$: denotes a hash function. It takes as input an arbitrary-length bit-string X and outputs a bit-string of some fixed length Y.

  For all implementations in this lab, we will exclusively be using either SHA256 or SHA384, imported from PyCryptodome **Crypto.Hash** [2].

- $k' \leftarrow \mathsf{KDF}(r, s, c, L)$ is a key derivation function. It takes as input some randomness r, some (optional) salt s, some context input c and an output length L, and outputs a key k'.

  In this lab, we will be using a key derivation function KDF (specifically, HKDF, which is implemented in `tls_crypto` using HMAC from **Crypto.Hash** [2]) to derive the symmetric keying material as well the output shared symmetric keys.

- $(\mathsf{sk}, \mathsf{vk}) \leftarrow_R \mathsf{SIG.KeyGen}(\lambda)$: denotes the key generation algorithm of a digital signature scheme SIG. It takes as input a security parameter λ, and outputs a signing key sk and a verification key vk.
- $\sigma \leftarrow_R \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})$: denotes the signing algorithm of SIG. It takes as input a signing key sk and a message msg, and outputs a signature σ.
- $\{0, 1\} \leftarrow \mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \sigma)$: denotes the verification algorithm of SIG. It takes as input a verifying key vk, a message msg, and a signature σ and outputs 0 or 1.

  Again, for correctness we need that for all $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{SIG.KeyGen}(\lambda)$, we have:

  $$\mathsf{SIG.Verify}(\mathsf{vk}, \mathsf{msg}, \mathsf{SIG.Sign}(\mathsf{sk}, \mathsf{msg})) = 1.$$

  In this lab, we will be using ECDSA [3] or RSA [4] signatures to instantiate our digital signature scheme, imported from **Crypto.Signatures**.

- $\tau \leftarrow \mathsf{MAC}(k, \mathsf{msg})$: denotes the message authentication algorithm MAC. It takes as input a symmetric key k and a message msg, and outputs a MAC tag τ.

  In this lab, we will be using HMAC to instantiate our MAC algorithm, imported from **Crypto.Hash** [2].

- $k \leftarrow_R \mathsf{AEAD.KeyGen}(\lambda)$: denotes the key generation algorithm of an authenticated encryption scheme with associated data AEAD. It takes as input a security parameter λ and outputs a symmetric key k.
- $\mathsf{ctxt} \leftarrow \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg})$: denotes the encryption algorithm of AEAD. It takes as input a symmetric key k, a nonce *nonce*, some associated data *ad* and a message msg, and outputs a ciphertext ctxt
- $\{\mathsf{ctxt}, \perp\} \leftarrow_R \mathsf{AEAD.Dec}(k, nonce, ad, \mathsf{ctxt})$: denotes the decryption algorithm of AEAD. It takes as input a symmetric key k, a nonce *nonce*, some associated data *ad* and a ciphertext ctxt, and outputs either a plaintext message msg or an error symbol $\perp$.

  For correctness we need that for all $k \leftarrow \mathsf{AEAD.KeyGen}(\lambda)$, all nonces *nonce*, for all associated data *ad* and all messages msg, we have:

  $$\mathsf{msg} = \mathsf{AEAD.Dec}(k, nonce, ad, \mathsf{AEAD.Enc}(k, nonce, ad, \mathsf{msg}))$$

In this lab, we will be using `AES_128_GCM` [4], `AES_256_GCM` [4] and `CHACHA20_POLY1305` [5] to instantiate our AEAD schemes, imported from **Crypto.Cipher** [4].

Limitations

Unfortunately, you will not be working on a full TLS 1.3 implementation, but a significantly streamlined and "bare-bones" variant of TLS 1.3. This implementation we have provided diverges from the specification in a number of ways (not an exhaustive list):

- We do not include the TLS alert protocol in our implementation. As a result, the server implementation will not send alert messages in response to some failure to parse a message or verify an authentication value. It will just close the connection.
- We do not include the majority of TLS extensions listed in the TLS 1.3 RFC, such as ServerNameIndication. We only use extensions for signature negotiation, ECDHE group negotiation, version negotiation, as well as the extensions required to implement session resumption and 0-RTT data.
- We do not include the use of ChangeCipherSpec, sent for compatibility purposes.
- Since we only exchange a single message between the client and server (and vice-versa) per handshake, we do not include KeyUpdate mechanisms within the Record Layer. Similarly, our implementation does not send Closure Alerts, which protect against truncation attacks.
- Since we know exactly which Diffie-Hellman groups that the server supports ahead of time, our server implementation does not support HelloRetryRequest in the event of failing to find sufficient information to proceed with a TLS 1.3 handshake.

The point that we are making here is that this is *not* a full TLS implementation, and we would not recommend its usage in the wild.

Part II: A Simplified Handshake

Implemented Functions

Before we begin, it is worth highlighting some code that has already been provided to you in the skeleton file. These are essentially functions based on the tinyEC library to implement elliptic curve cryptography [7]. These would assist you in generating a (scalar, point) key-pair, and execute basic scalar multiplication operations on the curve. If you are unfamiliar with last week's lab, you may wish to read the documentation [7], or refer to the Additional Listings section at the end of the sheet for an example of how to utilise tinyec.

High-Level API

Let's begin by looking at the highest level of our TLS implementation. This is not really a part of TLS, but instead a simple sockets implementation that allows network communication. The TLS specific parts of these functions is handled mostly transparently to the user.

In what follows, we will be focusing on implementing client-specific functions that are called during the Handshake, since that is what you will be implementing for your assessment. Below we give a listing for the "simple_client.py" file, and discuss what this does.

Listing 1: Simple Client Socket

```
import socket
from tls_application import TLSConnection

  def client_socket():
  s = socket.socket()
  host = socket.gethostname()
  #host = '18.216.1.168'
  port = 1189
  s.connect((host, port))
  client = TLSConnection(s)
  client.connect()
  client.write("challenge".encode())
  msg = client.read()
  print(msg.decode('utf-8'))
  s.close()
if __name__ == '__main__':
  client_socket()
```

This is a fairly simple function, and we go through how it works. The `client_socket` function begins by creating a socket object, allowing network communication. Afterwards, the `simple_client` function connects to the (*host*, *port*) pair provided and intialises a TLSConnection. TLSConnection uses a similar, but much simplified, as openssl's connection interface, i.e. the client calls `connect()` to initiate the handshake.

After a successful handshake the client can then simply send and read messages via TLSConnection's `write()` and `read()` calls.

Now we have seen how this API will be used, let us focus on the main tasks that you will have to complete for this lab.

Overview

In this lab, you will be implementing a series of TLS cryptographic primitives. You will have access to a folder containing a series of python files implementing various aspects of the TLS 1.3 protocol, and test vectors for testing your implementation. We list these below and describe on a high-level what each file is contributing to our TLS implementation:

- `simple_client.py`: This file creates sockets and manages networking tasks for a client TLS instance.
- `simple_server.py`: This file creates sockets and manages networking tasks for a server TLS instance.
- `test_tls_crypto.py`: This file will allow you to run unit tests and see how well your implementation of the tls-specific cryptographic primitives match ours
- `test_tls_handshake.py`: This file will allow you to run unit tests and see how well your implementation of the tls-specific client handshake functions match ours
- `tls_application`: This file contains the API that connects the high-level functions contained in simple_client.py and simple_server.py to the appropriate Handshake and Record functions contained in tls_handshake and tls_record_layer, respectively.
- `tls_crypto`: This file contains the tls-specific cryptographic functions. You should have implemented this yourself last week.
- `tls_error`: This file contains some basic errors that may occur during the execution of a TLS Handshake or TLS Record Layer protocol.
- `tls_extensions`: This file contains functions to manage the preparation and negotiation of TLS extensions sent during the ClientHello and ServerHello messages.
- `tls_handshake`: This file contains handshake functions for both client and server handshake functions, some of which you will be implementing for this assessment.
- `tls_psk_handshake`: This file contains and extended version of the Handshake with added PSK functionality that you will be implementing for both clients and servers.
- `tls_record_layer`: This file contains high-level API for preparing both plaintext and encrypted TLS record packets.
- `tls_state_machines`: This file contains the simplified TLS Handshake state machines.
- `tls_psk_state_machines`: This file contains the simplified TLS Handshake state machines. You will be extending the state machines in this file to support session resumption and 0-RTT data.
- `psk_client.py`: This file creates sockets and manages networking tasks for a client TLS instance. After a successful Handshake it tries to resume the session using PSKs and sends early data.
- `psk_server.py`: This file creates sockets and manages networking tasks for a server TLS instance. It supports PSKs and early data.

We build upon the cryptographic functions you implemented in the TLS part I portion of last week's lab to implement the Handshake protocol in this part and then add support for TLS session resumption and 0-RTT in part III below At the end of this lab all your changes will be in the `tls_handshake.py`, `tls_psk_handshake.py`, and `tls_psk_state_machines.py`, which together with `tls_crypto.py` from last week should include all your work on TLS.

In what follows, you will be expected to be able to support the following cryptographic options:

- Ciphersuites: `TLS_AES_128_GCM_SHA256`, `TLS_AES_256_GCM_SHA384`, `TLS_CHACHA20_POLY1305_SHA256`. This means that when you use hash functions or AEAD schemes, you will need to be able to distinguish between use of SHA256 or SHA384, and AES_128_GCM, AES_256_GCM, or CHACHA20_POLY1305 respectively. All functions that you implement that requires this distinct behaviour will be given `csuite` as input – an integer representation of the negotiated ciphersuite, which will allow you to distinguish which algorithms you require. The various `csuite` values are defined in `tls_constants.py`, and we recommend you look through this file.
- Elliptic-Curve Diffie-Hellman (ECDH) groups: You will required to support SECP256R1, SECP384R1 or SECP521R1. Similarly, all functions that you implement that requires distinct behaviour depending on the negotiated group will be given `neg_group` as input – an integer representation of the negotiated group. The various `neg_group` values are defined in `tls_constants.py`.
- Signature schemes: You will be required to support `RSA_PKCS1_SHA256`, `RSA_PKCS1_SHA384`, and `ECDSA_SECP384R1_SHA384`. As before, all functions that you implement that requires distinct behaviour depending on the negotiated signature scheme will be given `signature_algorithm` as input – an integer representation of the negotiated signature scheme. The various `neg_group` values are defined in `tls_constants.py`.

With our overview done, let us take a closer look at `tls_handshake.py`, and describe the expected API and operations for each.

TLS Handshake Functions

In this portion of the lab you will be implementing a series of Client Handshake functions. Specifically, the function that creates the `ClientHello` message, the function that parses the `ServerHello` message, the function that verifies the `ServerCertificateVerify` message, and finally, the function that verifies the `ServerFinished` message. All these functions are defined in `tls_handshake.py`

Before we begin, we should point to what a TLSPlaintext packet looks like. Consider the following from Section 5.1 of the TLS 1.3 RFC:

```
struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;
```

type: The higher-level protocol used to process the enclosed fragment.

legacy_record_version: MUST be set to 0x0303 for all records generated by a TLS 1.3 implementation other than an initial `ClientHello` (i.e., one not generated after a

HelloRetryRequest), where it MAY also be 0x0301 for compatibility purposes. This field is deprecated and MUST be ignored for all purposes. Previous versions of TLS would use other values in this field under some circumstances.

length: The length (in bytes) of the following TLSPlaintext.fragment. The length MUST NOT exceed 2^{14} bytes. An endpoint that receives a record that exceeds this length MUST terminate the connection with a "record_overflow" alert.

fragment: The data being transmitted. This value is transparent and is treated as an independent block to be dealt with by the higher- level protocol specified by the type field.

Note that ContentType is a integer represented as a single byte in big-endian (network) order:

```
enum {invalid(0), change_cipher_spec(20), alert(21),
       handshake(22), application_data(23), (255)} ContentType;
```

In `tls_handshake.py`, we define a function `attach_handshake_header` that, given an integer `msg_type` and a series of bytes `msg`, will concatenate this handshake header to the beginning of `msg`. Similarly, we have defined a function `process_handshake_header` that, given an integer `msg_type` and a series of bytes `msg`, strips the header from the message `msg`. If these messages are sent encrypted, the plaintext is then given another wrapper:

```
struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;
```

Finally, the RecordLayer header is then attached to the message:

```
struct {
    ContentType opaque_type = application_data; /* 23 */
    ProtocolVersion legacy_record_version = 0x0303; /* TLS v1.2 */
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

You may recall the second part from the AD field used in the AEAD scheme you implemented in part I of this lab. While interesting, the Record Layer is out of scope for your assessment, so let's circle back around to the function `tls_13_client_hello`.

TLS Client Hello

Listing 2: TLS Client Hello

```
def tls_13_client_hello(self):
  raise NotImplementedError()
```

In this function, you will be implementing the creation of a valid TLSPlaintext `ClientHello` message. This function should take no additional inputs (beyond *self*) and output `client_hello_msg` – a series of bytes comprising the ClientHello message. Section 4.2.1 of the TLS 1.3 RFC describes the structure of a TLS ClientHello message, consider the following below:

Structure of this message:

```
struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;

uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];    /* Cryptographic suite selector */
```

Recall that the $< X..Y >$ notation means that the field must be preceded by a length-encoding, large enough to represent Y in big-endian (network) order. The `Random` and `legacy_session_id` structures are straightforward – according to the TLS 1.3 RFC, both are 32-bytes of randomness. The `Random` structure acts as a nonce, so it should be generated via a "secure random number generator". The handshake object is intialized with a secure random number generator in **self.get_random_bytes** for you to use.

Ciphersuites is a list of the symmetric cipher options supported by the client – the values for the ciphersuites you are expected to support are given in `tls_constants.py`. Each "ciphersuite" in this list is a 2-byte representation of the integer value given in `tls_constants.py`, in big-endian order. To make this simpler for you, we have already initialised the client and server with the list of ciphersuites they will support, in *self*.`csuites` (where *self*.`csuites` is a tuple of integers). you will have to encode each integer as a 2-byte big-endian representation. Don't forget to add the length encoding!

The `legacy_compression_methods` "MUST contain exactly one byte, set to zero, which corresponds to the "null" compression method in prior versions of TLS." Remember that the $<>$ notation means that you must still include the length-encoding field.

In our TLS implementation, we support the following extensions:

- `supported_versions`
- `supported_groups`
- `key_share`
- `signature_algorithms`

For the structure of the extension fields, consider the following from Section 4.2 of the TLS 1.3 RFC:

```
struct {
     ExtensionType extension_type;
     opaque extension_data<0..2^16-1>;
} Extension;
```

Each extension, like a Handshake or RecordLayer header, begins with a `extension_type` identifying the extension, and a field encoding the length of the `extension_data`.

The file `tls_extensions.py` contains functions for preparing each extension:

- `tls_extensions.prep_support_vers_ext(extensions)`
- `tls_extensions.prep_support_groups_ext(extensions)`
- `tls_extensions.prep_keyshare_ext(extensions)`
- `tls_extensions.prep_signature_ext(extensions)`

Each takes *self*.`extensions` as input (which we have already initialised in our implementation), and outputs the given extension in byte format.

The only exception is `tls_extensions.prep_keyshare_ext`, which will return two outputs: the extension itself, and a dictionary of ECDH secret values, indexed by the integer representation of each supported group. You should save this dictionary to *self*.`ec_sec_keys`, as you will need to extract one of these later in the handshake. We recommend you look through the `tls_extensions.py` file to determine what each of these extensions look like, and how negotiation of such extensions occur.

In standard TLS implementations, the ordering of extensions should not matter. To ensure easier testing of your implementation we specify the following order:

1. Supported Version
2. Supported Group
3. Keyshare
4. Signature Algorithm

Now you have created each of the fields in the `ClientHello`, simply concatenate the fields (in the correct order!) and add the appropriate TLSPlaintext header. Your implementation will also need to update *self*.`transcript` to include the `ClientHello` message, in order to compute the transcript hash in later stages of the Handshake.

This portion of the assessment is intended to familiarise you with TLS packet structure, reading specifications and correctly aligning expected inputs for various pre-established functions. Now we have finished discussing how to create a ClientHello message, let's continue onto the second part of our Client Handshake functions for you to implement: `tls_13_process_server_hello`.

TLS Process Server Hello

Listing 3: TLS Process Server Hello

```
def tls_13_process_server_hello(self, shelo_msg):
  raise NotImplementedError()
```

As the name suggests, in `tls_13_process_server_hello` you will be implementing a function that parses the ServerHello message, and extracts the *ciphersuite*, *version*, *ECDH group* and *ECDH keyshare* that the server has negotiated. This means implementing code that can parse variable-length extensions. In addition, your function will also use previously established API to compute secret ECDH values, and derive a series of secrets. This function should take inputs *self*, `shelo_msg` (where *self* is the current state, and `shelo_msg` a series of byte comprising the ServerHello message), and return no output. Section 4.1.3 of the TLS 1.3 RFC describes the structure of a TLS ServerHello message:

```
Structure of this message:

struct {
    ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;
```

Recall again that the $< X..Y >$ notation means that the field must be preceded by a length-encoding, large enough to represent Y in big-endian (network) order. The `Random` is generated in exactly the same way as in `ClientHello` – by generating 32 random bytes. However, `legacy_session_id` should by an echo of the client's recently sent `legacy_session_id` – you can check for yourself that our implementation does this in `tls_13_process_client_hello` and `tls_13_server_hello`.

After the `legacy_session_id` comes the single CipherSuite that the server has negotiated – your implementation should set *self*.`csuite` to the *integer* value of this field. `legacy_compression_method` is a single byte that indicates that the server's choice of compression method. In TLS 1.3 this must be the "null" compression method – hence `legacy_compression_method` = 0.

Finally, you must process the extensions that the Server has sent. These will be following extensions:

- `supported_versions`
- `key_share`

You'll notice that one of the extensions sent in the `ClientHello` is not present, specifically `signature_algorithms`. Why would the Server not need to send this extension?

As with CipherSuite, `supported_versions` will contain a single Version that the Server has negotiated. *Unlike* CipherSuite, you will need to parse the extension identifier and extension length. Your implementation should set *self*.`neg_version` to the *integer* value of this field.

The `key_share` extension has a slightly more complex format, which we'll examine below. Consider the following from Section 4.2.8 (KeyShare) from the TLS 1.3 RFC:

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

Keep in mind that this `KeyShareEntry` is the `extension_data` sent in an `Extension`. `KeyShareEntry` contains the `NamedGroup` and the confusingly titled `key_exchange` fields. `NamedGroup` is a 2-byte representation of integer value indicating the NamedGroup, defined in the TLS 1.3 RFC. For our implementation, these integer values are defined in `tls_constants.py`, for instance `SECP256R1_VALUE` $=$ 0x0017. Here `key_exchange` is the Diffie-Hellman keyshare. First, note the variable-length notation for `key_exchange`. Next, we turn to the TLS 1.3 RFC to see how the `key_exchange` field is formatted, Section 4.2.8 :

```
struct {
    uint8 legacy_form = 4;
    opaque X[coordinate_length];
    opaque Y[coordinate_length];
} UncompressedPointRepresentation;
```

Our TLS implementation follows this format: A single byte, followed by the X and Y co-ordinates of the Elliptic Curve Diffie-Hellman key share. To make this simpler, we have included two functions in `tls_crypto.py` to help you convert between tinyEC elliptic curve Point objects and bytes:

- `convert_ec_pub_bytes(ec_pub_key, group_name)`
- `convert_x_y_bytes_ec_pub(pub_bytes, group_name)`

Straightforwardly, `convert_ec_pub_bytes(ec_pub_key, group_name)` takes as input a tinyEC EC Point object, and a group name and return a series of bytes.

`convert_x_y_bytes_ec_pub(pub_bytes, group_name)` takes a series of bytes and a group name, and returns a tinyEC Point object. For consistency, the `group_name` is the integer value used to indicate `NamedGroups` in `KeyShare` extensions.

Your implementation, when parsing `KeyShareEntry`, should convert the `NamedGroup` to an integer, and then use `tls_crypto.convert_x_y_bytes_ec_pub` to create a tinyEC Point object. After, the function should set *self*.`ec_pub_key` to this tinyEC Point object.

Once you have finished processing the `ServerHello` message, there are only three parts left to complete for this function:

1. Update *self*.`transcript`, by concatenating the `shelo_msg` to the end.
2. Compute the Diffie-Hellman secret value, using:
   - `tls_crypto.ec_dh(ec_sec_key, ec_pub_key)`, which takes as input an integer `ec_sec_key` and a tinyEC Point `ec_sec_key`, and returns a tinyEC Point `ec_secret_point`. For this part, you will need to extract the ECDH secret value from *self*.`ec_sec_keys`, set during `tls_13_client_hello`. Recall what type of structure *self*.`ec_sec_keys`, and how to extract values from that structure. You can examine `tls_constants.py` `GROUP_FLAGS` to see how we convert integers used in TLS to indicate groups, to strings used in tinyEC to indicate groups, in order to make the interface on the `tls_handshake.py` level more uniform for you.
   - `tls_crypto.point_to_secret(ec_point, group)`, which takes as input a tinyEC Point `ec_point`, and an integer `group` and returns a series of bytes `ecdh_secret`
3. Extract and derive these secrets, according to the key schedule on the next page:
   - *self*.`early_secret`, using `tls_crypto.tls_extract_secret`.
   - *self*.`handshake_secret`, using `tls_crypto.tls_derive_secret` and `tls_crypto.tls_extract_secret`.
   - *self*.`client_hs_traffic_secret`, using `tls_crypto.tls_derive_secret`
   - *self*.`server_hs_traffic_secret`, using `tls_crypto.tls_derive_secret`
   - *self*.`master_secret`, using `tls_crypto.tls_derive_secret` and `tls_crypto.tls_extract_secret`.

When **None** appears in the key schedule, use this input literally, i.e. `early_secret` = `tls_crypto.tls_extract_secret`(*self*.csuite, None, None).

```
          None
           |
           v
None   ->  HKDF-Extract = Early Secret
           |
           +-----> Derive-Secret(., "ext binder" | "res binder", "")
           |                     = binder_key
           |
           +-----> Derive-Secret(., "c e traffic", ClientHello)
           |                     = client_early_traffic_secret
           |
           +-----> Derive-Secret(., "e exp master", ClientHello)
           |                     = early_exporter_master_secret
           v
           Derive-Secret(., "derived", "")
           |
           v
(EC)DHE -> HKDF-Extract = Handshake Secret
           |
           +-----> Derive-Secret(., "c hs traffic",
           |                     ClientHello...ServerHello)
           |                     = client_handshake_traffic_secret
           |
           +-----> Derive-Secret(., "s hs traffic",
           |                     ClientHello...ServerHello)
           |                     = server_handshake_traffic_secret
           v
           Derive-Secret(., "derived", "")
           |
           v
None    -> HKDF-Extract = Master Secret
           |
           +-----> Derive-Secret(., "c ap traffic",
           |                     ClientHello...server Finished)
           |                     = client_application_traffic_secret_0
           |
           +-----> Derive-Secret(., "s ap traffic",
           |                     ClientHello...server Finished)
           |                     = server_application_traffic_secret_0
           |
           +-----> Derive-Secret(., "exp master",
           |                     ClientHello...server Finished)
           |                     = exporter_master_secret
           |
           +-----> Derive-Secret(., "res master",
                                 ClientHello...client Finished)
                                 = resumption_master_secret
```

TLS Process Server Certificate Verify

Listing 4: TLS Process Server Certificate Verify

```
def tls_13_process_server_cert_verify(self, verify_msg):
    raise NotImplementedError()
```

In this function, you will be processing the first server authentication message, ServerCertificateVerify. On a high-level, the message is simply a signature over a hash of the current transcript, which you will verify using the server public-key that can be extracted from the ServerCertificate message. This function should take inputs *self*, `verify_msg` (where *self* is the current state, and `verify_msg` a series of bytes comprising the ServerCertificateVerify message, and return no output.

Much like previous messages, you will first need to process the handshake header via *self*.`process_handshake_header`, which takes as input an integer representing the expected handshake (see `tls_constants` for definitions of handshake types) and the handshake message itself (given in bytes).

Consider the following from Section 4.4.3 (Certificate Verify) of the TLS 1.3 RFC:

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

`SignatureScheme algorithm` here refers to the 2-byte representation of the integer value corresponding to the Signature scheme that the Server used to create the signature. You can see now why the Server did not need to send an extension indicating which signature scheme was negotiated, as it is indicated within this CertificateVerify message instead. Again, take note of the variable-length notation here, and recall what that implies for the structure of the message. Once your implementation has extracted the signature from the CertificateVerify message, there are only four steps that follow:

1. Extract the public-key from the certificate that the Server sent in an earlier message. To help you, our implementation has saved the Server Certificate as a string in *self*.`server_cert_string`. In addition, we have provided the following functions:
    - `tls_crypto.get_rsa_pk_from_cert`, which takes as input a certificate string, and outputs an RSA Public Key object
    - `tls_crypto.get_ecdsa_pk_from_cert`, which takes as input a certificate string, and outputs an ECDSA Public Key object
2. Create a transcript hash from the currently maintained *self*.`transcript`. Your previous implementation of `tls_transcript_hash` will help you here.
3. Verify the signature. Your previous implementation of `tls_verify_signature` will help you here.

4. If the signature verifies correctly, update *self*.`transcript` with the CertificateVerify message you just processed.

TLS Process Finished

Listing 5: TLS Process Finished

```
def tls_13_process_finished(self, fin_msg)
  raise NotImplementedError()
```

In this function, you will be processing the second authentication message, the Finished message. Your implementation should be general enough for both the client and server to use this function. On a high-level, the message is simply a MAC tag over a hash of the current transcript, which was computed using the `finished_key`.

Much like previous messages, you will first need to process the handshake header via *self*.`process_handshake_header`, which takes as input an integer representing the expected handshake (see `tls_constants` for definitions of handshake types) and the handshake message itself (given in bytes).

Consider the following from Section 4.4.4 (Finished) of the TLS 1.3 RFC:

> Structure of this message:
>
> ```
> struct {
>     opaque verify_data[Hash.length];
> } Finished;
> ```

That's it! Once you have stripped the handshake header from the message, all that's left is the MAC tag itself. Thus, all you must do in this function is:

1. Derive the `finished_key`. Consider the following from the TLS 1.3 RFC:

   - `finished_key` = HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)

   Your implementation of `tls_finished_key_derive` will help you here. To save you searching through the TLS 1.3 RFC (the definition of BaseKey is very well-hidden), BaseKey here is the `server_hs_traffic_secret`, which you saved into the state in a previous function!

2. Create a transcript hash from the currently maintained *self*.`transcript`. Your previous implementation of `tls_transcript_hash` will help you here.

3. Verify the MAC tag. Your previous implementation of `tls_finished_mac_verify` will help you here.

4. If the signature verifies correctly, update *self*.`transcript` with the CertificateVerify message you just processed.

5. If *self*.`role` is client, then you will also need to compute *self*.`client_ap_traffic_secret`, *self*.`server_ap_traffic_secret`. You will need to compute a new transcript hash for this step, since you just updated the *self*.`transcript`. Your previous implementation of `tls_derive_secret` will help you here.

Part III: Session Resumption and 0-RTT

This part can be conceptually divided in two sub-parts. First we will be implementing a series of TLS PSK functions that are used during session resumption to create and parse extensions. Then we will make changes to the state machine and the remaining Handshake functions so that our server and client actually make use of session resumption and early data. Please note that by now we assume some familiarity with the code base, the TLS 1.3 RFC, and its presentation language (e.g. when to add length encoding). So we will expect that you look up missing details in the TLS 1.3 RFC.

We start with taking a closer look at `tls_psk_handshake.py`, and describe the expected API and operations for each function. PSKHandshake defined here is inheriting from class Handshake in `tls_handshake.py`. The idea behind this is to reduce code duplication for functions that stay the same, and allow easier code reuse for functions where you might only need to add a few extra steps at the end. This should also prevent you from accidentally breaking your existing solution for the first part of the lab.

Do *not* make any further changes in `tls_handshake.py` other than the ones described in the previous sections. Everything that follows here should happen in `tls_psk_handshake.py` and `tls_psk_state_machines.py`.

Listing 6: PSKHandshake initialtization

```
def __init__(self, csuites: List[int], extensions: Dict[int, List[int]],
             role: int, psks: List[Dict[str, Union[bytes, int]]] = None,
             psk_modes: List[int] = None, server_static_enc_key: bytes = None,
             early_data: bytes = None):
  super().__init__(csuites, extensions, role)
  self.psks = psks
  self.psk = None
  self.psk_modes = psk_modes
  self.server_static_enc_key = server_static_enc_key
  self.early_data = early_data
  self.client_early_traffic_secret = None
  self.accept_early_data = False
  self.selected_identity = None
  self.resumption_master_secret = None
  self.max_early_data = None
  self.offered_psks = None
  self.use_keyshare = None
  self.client_early_data = None
  self.get_time = timer
  self.get_random_bytes = get_random_bytes
```

Here you can see the init method of PSKHandshake. Together with the init function of Handshake it already initializes all fields that are necessary. Whenever you find that you need a value that you are not given as parameter and cannot compute it is likely that the function you are implementing is only called in states where these fields should have been set to values you need. If you get confused about the state you should currently be in have a look at the TLS 1.3 RFC. Figure 3 in Section 2.2 of the RFC gives an overview of how handshakes with session resumption using a PSK look.

Server NewSessionTicket

Listing 7: Server NewSessionTicket

```
def tls_13_server_new_session_ticket(self):
  raise NotImplementedError
```

In this task, you will be implementing a function that creates a post-handshake message known as the `NewSessionTicket` message, described in Section 4.6.1 of the TLS 1.3 RFC. According to the RFC, the `NewSessionTicket` message has the following structure:

```
struct {
  uint32 ticket_lifetime;
  uint32 ticket_age_add;
  opaque ticket_nonce<0..255>;
  opaque ticket<1..2^16-1>;
  Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

You should set these values according to the following instructions:

- `ticket_lifetime` is the validity period of the use of the PSK. Set this value to the maximum lifetime of tickets as instructed in the specification, which is 604800 seconds, and given in seconds.
- `ticket_age_add` is a randomly generated value, used to obscure the actual age of PSKs when sent in `PreSharedKeyExtensions`.
- `ticket_nonce` is a randomly generated value, that is used to generate the PSK. We mandate 8 bytes for the `ticket_nonce`.
- `ticket` is an encrypted PSK, along with enough information for the server to later verify that a PSK is still valid. The specification states that "[the ticket] MAY be either a database lookup key or a self-encrypted and self-authenticated value." Our implementation will take the latter approach. We discuss how the `ticket` value is generated below.
- `extensions` is a single extension: `Early Data Indication`, which allows the server to specify the maximum number of bytes the client will be allowed to send in a 0-RTT connection. In our implementation, we have chosen 2^{12} as the maximum number of bytes that the server will allow the client to send. You will need to construct an extension according to the following specification, found in Section 4.2.10 of the TLS 1.3 RFC:

  ```
  struct {
    select (Handshake.msg_type) {
      case new_session_ticket:   uint32 max_early_data_size;
      case client_hello:         Empty;
      case encrypted_extensions: Empty;
  ```

```
  };
} EarlyDataIndication;
```

Here, `uint32 max_early_data_size` is a 4-byte representation of the maximum number of bytes that the client is allowed to send in network (big-endian) order.

The ticket allows the server to recover the PSK from the ticket, when the client returns the ticket in a `PreSharedKeyExtension`. We will be using a common technique that allows for stateless servers to recover necessary information: Session Ticket Encryption. We specify here that the server will always use ChaCha20_Poly1305 to encrypt their tickets. This is an AEAD scheme, so recall that $\mathsf{AEAD.Enc}(k, N, ad, ptxt)$ takes four inputs: a key k, which is stored in the state as `server_static_enc_key`, a nonce N that will be randomly generated by your implementation, associated data ad (which we will not be using for the encrypted ticket, so there is no need to update the cipher with an ad), and a plaintext $ptxt$. We need to include enough information for the server to recover the PSK, and verify that the age of the PSK is not greater than the `ticket_lifetime`, so we specify that the plaintext should be:

$$ptxt = PSK \| \texttt{ticket\_add\_age} \| \texttt{ticket\_lifetime} \| \texttt{csuite}$$

You can assume that `csuite` is stored in the state (i.e. in *self*.`csuite`). To compute the PSK from the `ticket_nonce` and the resumption master secret, consider the following from Section 4.6.1 of the TLS 1.3 RFC:

The PSK associated with the ticket is computed as:

```
HKDF-Expand-Label(resumption_master_secret, “resumption”, ticket_nonce,
Hash.length)
```

You may wish to use the TLS-specific cryptographic primitives you implemented in `tls_crypto.py`. You can assume that the PSKHandshake object already has the resumption master secret stored in its state.

The addition of the `csuite` allows the server to check that the PSK was derived with a consistent hash function – a requirement of the specification. Encrypt the plaintext, and construct the ticket as the concatenation of the nonce N, the ciphertext $ctxt$ and the output tag τ. Afterwards, construct the `NewSessionTicket` message as indicated above. Your function should return the `NewSessionTicket` message in byte format.

To enable testing of this function despite the randomness being sampled, you have to use the random number generator stored in the PSKHandshake object, i.e. `self.get_random_bytes`. During testing and evaluation we will overwrite this random number generator to produce predictable numbers.

Client Parse NewSessionTicket

Listing 8: Client Parse NewSessionTicket

```
def tls_13_client_parse_new_session_ticket(self, nst_msg):
  raise NotImplementedError
```

This function should take as input a `NewSessionTicket` message, both in byte-format. The function should parse the `NewSessionTicket` message, derive a PSK, an early secret, and a binder key, according to the key schedule, given in the Appendices.

It should construct a PSK dictionary object which, given a key X should return a value Y according to the table below. Parsing the `NewSessionTicket` message and deriving the secrets according to the key schedule should allow you to compute all values stated below.

Key X	Value Y
"PSK"	psk
"lifetime"	`ticket_lifetime`
"lifetime_add"	`ticket_add_age`
"ticket"	`ticket`
"max_data"	`max_data`
"binder key"	`binder_key`
"csuite"	*self*.`csuite`
"arrival"	arrival time

The function should return the PSK dictionary object. You can assume that the *self* state has already been initialised with the appropriate ciphersuite and resumption_master_secret. We provide a function to get the current time. You can access this function via `self.get_time`; it simply return the current time in milliseconds. Make sure to use this function and access it via `self.get_time`. **If you do not use this function the tests will likely fail and you might not get any points for this task.**

Client Prepare PskModeExtension

Listing 9: Client Parse NewSessionTicket

```
def tls_13_client_prep_psk_mode_extension(self):
  raise NotImplementedError
```

In this function you will create a PSK extension, which indicates to the server which PSK modes the client supports. You can assume that the state of contains a tuple of integers, `psk_modes`, indicating the PSK modes that it supports, as described in Section 4.2.9:

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeMode;

struct {
  PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

Finally, the function should output a valid `PskKeyExchangeModes` extension in byte format.

Client Add PSK Extension

Listing 10: Client Add PSK Extension

```
def tls_13_client_add_psk_extension(self, chelo, extensions):
  raise NotImplementedError
```

In this function, you will create a PSK extension, which will indicate to the server the list of `OfferedPsks` and identities that the client is willing to support. The extension will take as input: `chelo` the client helo as bytes object corresponding to the CLientHello struct in Section 4.1.2 of the TLS 1.3 RFC, but without the extensions field, as well as `extensions` a bytes object containing all the extensions the client will add to the client hello before adding the PSK extension.

Your implementation should use all PSK dictionaries in the state to generate a `PreSharedKeyExtension` according to the specification described in Section 4.2.11 of the TLS 1.3 RFC. The `extension_data` field of this extension contains a `PreSharedKeyExtension`. See the following details below:

```
struct {
  opaque identity<1..2^16-1>;
  uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
  PskIdentity identities<7..2^16-1>;
  PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
  select (Handshake.msg_type) {
    case client_hello: OfferedPsks;
    case server_hello: uint16 selected_identity;
  };
} PreSharedKeyExtension;
```

```
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^16-1>;
} Extension;
```

It will be useful to note that thePreSharedKeyExtension is the input extension_data here. Note here that identity<1..2^{16}-1> corresponds to the ticket value saved in PSK dictionary, and obfuscated_ticket_age is a representation of the ticket age added to the "lifetime_add" value saved in the PSK dictionary, modulo 2^{32}. This is described in Section 4.2.11.1 of the TLS 1.3 RFC.

You should compute the ticket age using the point in time you enter the function and the arrival time of the ticket. Again, use the provided function to get the current time. You can access this function via self.get_time; it simply return the current time milliseconds. Make sure to use this function and access it via self.get_time. **If you do not use this function the tests will likely fail and you might not get any points for this task.**

The binder value forms a binding between the PSK, the current handshake, and the handshake from which it was generated. Each entry in the binders list is computed as a HMAC over the transcript hash, containing a ClientHello message and including extensions up to the PreSharedKeyExtension.identities field. The length fields for the messages (including the overall length, the length of the extension block, and the length of the PSK extension) are set as if the binders of the correct lengths are present. You have to construct this truncated transcript using the chelo and the extensions that the function takes as input.

Thus, you will need to compute the identities list before you can compute the binders themselves, and then determine what is the expected full length of the extension. Each PSK dictionary contains the integer representation of the csuite you will compute each binder with, and that, paired with hash.digest_size(), will allow you to precompute the expected lengths of the binders list. Don't forget to include the length encoding of the binders list into your expected extension length.

PskBinderEntry is computed as the Finished message, but with the BaseKey being the binder_key included in the PSK dictionary.

Once you have the list of binder values, create the final PreSharedKeyExtension, and return the PreSharedKeyExtension in byte format as well as a list containing all the psk dictionaries that the client offered.

Server Prepare PSK Extension

Listing 11: Server Parse PSK Extension

```
def tls_13_server_parse_psk_extension(self, psk_extension)
  raise NotImplementedError
```

In this function, you will parse the `PreSharedKeyExtension`, and iteratively use the `server_static_enc_key` to sequentially decrypt `tickets`, and recover the PSK, `ticket_add_age` and `ticket_lifetime` values. Then, you will use the PSK to generate a binder key. You must verify that the `csuite` indicated in the `ticket` matches the server's negotiated ciphersuite (contained in *self*.`csuite`). Finally, that the binder value verifies, a truncated transcript as during the binder creation above. You can assume that the PSK extension is always the last extension in the Client Hello msg.

Hint: If you think about the state you are currently in, you might realize that computing the truncated transcript is much easier now than it was for the client generating the extension

The function should return the first such PSK and the index of the selected identity, where those conditions are met. Otherwise, the server should move onto the next `identity` and `binder` value in the extension. If no such `identity/binder` pairs are valid, the server should raise a `TLSError`.

Note that the TLS 1.3 RFC specifies that a binder verification failure should be a critical failure, you can indicate this via raising a `BinderVerificationError`.

A working Session Resumption

In the following section we will make sure our client and server can make use of our newly added PSK functionality. To this end we will have to extend the state machine implementation with the necessary state transition and make sure that our new PSK functions are called in the appropriate states of the handshake.

First we give an overview over the interface the client and server will use. We provide a psk_client.py as well as a psk_server.py which are extended versions of the simple_client.py and simple_server.py you used in the beginning of the lab. Both client and server pass an additional `use_psk=True` argument to the connect and accept calls respectively. This makes sure our TLSConnection will use the PSKHandshake object instead of the Handshake object. The server additionally also passes a static key, the server ticket encrypting key, to the accept call.

After a successful handshake, and a roundtrip of application messages, the client gets the PSKs from the TLSConnection, so that it can use them for subsequent connections. It then opens a new connection, this time adding the PSKs, psk modes and the early data as optional arguments.

The current implementation uses a simplified version of the TLS 1.3 state machine described in Appendix A of the TLS 1.3 RFC. Listings 12 and 13 show how we simplified the state machines, that are provided in tls_state_machines.py. Make sure you familiarize yourself with the state machines for the simplified handshake. To not encourage you to break the existing implementation you will make all your changes in tls_psk_state_machines.py, which currently is a copy of tls_state_machines.py with the exception being using a PSKHandshake object instead of a simple Handshake object.

Listing 12: Simplified Client State Machine

```
                  START
 Send ClientHello  |
                   |
                   v
                WAIT_SH
                   | Recv ServerHello
                   | K_recv = handshake
                   V
                WAIT_EE
                   | Recv EncryptedExtensions
                   |
                   |
                   v
            WAIT_CERT_CR
              Recv |
       Certificate |
                   |
                   |
                   v
                WAIT_CV
                   | Recv CertificateVerify
                   v
             WAIT_FINISHED
                   | Recv Finished
                   | K_send = handshake
 Can send          | Send Finished
 app data   -->    | K_send = K_recv = application
 after here        v
               CONNECTED
```

Listing 13: Simplified Server State Machine

```
                    START
     Recv ClientHello |
                      v
                  RECVD_CH
                      | Select parameters
                      v
                  NEGOTIATED
                      | Send ServerHello
                      | K_send = handshake
                      | Send EncryptedExtensions
Can send              | Send Certificate + CertificateVerify
app data              | Send Finished
after   -->           | K_send = application
here                  |
            No 0-RTT  |
                      |
   K_recv = handshake |
                      |
                      |
                      |
                      |
                      v
                 WAIT_FLIGHT2
                      |
                      |
             No auth  |
                      |
                      v
                WAIT_FINISHED
                      | Recv Finished
                      | K_recv = application
                      v
                  CONNECTED
```

Listings 14 and 15 show how we want the state machine to look for PSK and 0-RTT support. On the clients side we only have to make sure to be able to skip over the certificate states in the case the server accepts our PSK. Similarly for the server the only changes in the state machine are related to receiving 0-RTT data. For simplicity the server will always accept the PSK if the binder verifies. It will also always accept 0-RTT data as long as it picked the first PSK (index 0). We will start describing the changes to handle the PSK first, with the changes to support 0-RTT data following further below.

We suggest you also implement it in this order, i.e. make sure that session resumption with PSKs is working before trying to add 0-RTT on top. In addition to the changes in the state machine, we will also have to make changes to the following steps in the Handshake for the client:

1. If the client has any PSKs, it adds both a PSK mode extension as well as a PSK extension to the its Client Hello message. (Methods to construct these have already been implemented by you in the earlier parts of this lab.) Again to make testing easier the PSK mode and PSK extension should be added at the end of the list of extensions.
2. If we offered a valid PSK to the server, the server hello will contain a PSK Extension containing the PSK identity selected by the server in form of an index into the PSK list sent by the server.
3. Depending on the PSK modes we have sent to the server, the server will indicate the mode used by either sending its DHE keyshare or not. If it sends a keyshare we have to make sure that we compute a shared secret, if it does not we have to be able to handle this and only use the psk. Note that we have to make sure that DHE without PSK remains functional.
4. During processing of the server finished message the client needs to derive the resumption master secret, to enable him to process any new session ticket messages the server might send after the handshake.

To implement these changes you should make extended versions of `tls_13_client_hello`, `tls_13_process_finished` and `tls_13_process_server_hello` in **tls_psk_handshake.py**

Hint: Keep in mind that PSKHandshake inherits from Handshake, so all its helper functions are available. For some cases you might also be able to use inheritance to keep your code simple, although you can of course always just copy the functions you try to implement from Handshake to PSKHandshake and start from there instead.

For the server we need to make the following changes:

1. The Client hello can contain a PSK and a PSK mode extension, so we have to make sure `tls_13_server_get_remote_extensions` is able to handle this. Its return value is a dictionary containing the different extensions the client sent. It should use keys "psk" and "psk mode" for the PSK extension and the PSK mode extension respectively.
2. If a PSK and a PSK mode extension were sent by the client, `tls_13_server_select_parameters` needs to select the PSK. It also has to make sure that the `use_keyshare` field is set correctly depending on the use of PSKs and the PSK modes supported by the client. If the client supports PSK DHE the server should always pick doing the additional DHE for forward secrecy.
3. `tls_13_prep_server_hello` generates the server hello message. This needs to contain a keyshare if the Handshake is going to use DHE. If a psk was selected then we

have to send a PSK extension to tell the client which PSK we selected. At the end we have to make sure to derive the server_hs_traffic_secret, client_hs_traffic_secret, and the master_secret.

Listing 14: Simplified Client State Machine with 0-RTT

```
                             START
              Send ClientHello |
       [K_send = early data] |
                               v
        /                  WAIT_SH
        |                      | Recv ServerHello
        |                      | K_recv = handshake
    Can |                      V
   send |                  WAIT_EE
  early |                      | Recv EncryptedExtensions
   data |             +--------+--------+
        |       Using |                 | Using certificate
        |         PSK |                 v
        |             |            WAIT_CERT_CR
        |             |            Recv |
        |             |     Certificate |
        |             |                 |
        |             |                 |
        |             |                 v
        |             |              WAIT_CV
        |             |                 | Recv CertificateVerify
        |             +> WAIT_FINISHED <+
        |                      | Recv Finished
        \                      | [Send EndOfEarlyData]
                               | K_send = handshake
Can send                       | Send Finished
app data   -->                 | K_send = K_recv = application
after here                     v
                           CONNECTED
```

Listing 15: Simplified Server State Machine with 0-RTT

```
                                START
              Recv ClientHello |
                               v
                           RECVD_CH
                               | Select parameters
                               v
                           NEGOTIATED
                               | Send ServerHello
                               | K_send = handshake
                               | Send EncryptedExtensions
Can send                       | [Send Certificate + CertificateVerify]
app data                       | Send Finished
after   -->                    | K_send = application
here                  +--------+--------+
             No 0-RTT |                 | 0-RTT
                      |                 |
  K_recv = handshake  |                 | K_recv = early data
[Skip decrypt errors] |   +------> WAIT_EOED -+
                      |   |       Recv |       | Recv EndOfEarlyData
                      |   | early data |       | K_recv = handshake
                      |   +------------+       |
                      |                        |
                      +> WAIT_FLIGHT2 <--------+
                               |
                               |
                       No auth |
                               |
                               v
                         WAIT_FINISHED
                               | Recv Finished
                               | K_recv = application
                               v
                           CONNECTED
```

Early Data

Using 0-RTT (early data) allows the client to send encrypted application data immedietely following the client hello. For this the client offers PSKs to the server and assumes that the server will select the first (index 0) PSK. Early data is then encrypted under the client_early_traffic_secret (see Section 7.1 of the TLS 1.3 RFC for information on how to derive it).

To send early data the client adds an early data indication extension to its Client Hello msg (remeber that we require the PSK extensions to be the last extension in the Client Hello!). After sending that Client Hello it can then send early data as application messages. For simplicity we assume that our client only sends one TLSCiphertext of early data and does that immedietely after the client hello. If the server accepts early data, which in our case it should always do (given that it can select the correct PSK), it sends an EarlyDataIndication as part of the EncryptedExtensions message.

If the client received an EarlyDataIndication it should send end EndOfEarlyData message to the server after it received the Server's finished message. If it does not receive an EarlyDataIndication it has to handle that the server was not able to accept the early data. In our implementation we will just silently ignore this, but in a full implementation the user should be notified, so that they can judge whether it should be resend after a successful handshake. Note that the client is only allowed to send as much early data as the server specified via the max_early_data_size in the new session ticket.

For this last step we thus need to make the necessary changes to the state machine as well as `tls_13_client_hello`, `tls_13_server_get_remote_extensions`, `tls_13_server_select_parameters`, `tls_13_prep_server_hello`, and `tls_13_process_server_hello` like you did for the PSK support. In addition we now also need to make changes to `tls_13_server_enc_ext` and `tls_13_process_enc_ext` which previuosly only sent and parsed and empty EncryptedExtensions message.

Testing and Evaluation

We have provided in the skeleton file two test modules – all of which involve file reads and writes. You can use these modules to test your implementation. The test modules are summarized as follows:

1. The first module tests the correctness of the `tls_handshake.py` functions over various test vectors.
2. The second module tests the correctness of `tls_psk_handshake.py` functions over various test vectors.

For each of these test modules, you are provided with the input test vectors and the corresponding output vectors in separate input and output files. Running the test modules will tell you which functions likely output correctly. However the unit test provided by us do not cover all possible edge cases.

To run the test modules, simply run **python3 -m unittest** in the terminal, and the unit tests will print the results to the terminal.

Note: It might be a good idea to test your code using your own custom-designed test modules. However, we will be using test modules like the ones in the skeleton file to evaluate your submissions under an automated evaluation framework.

Evaluation

When we evaluate your submissions, we will run similar tests, however, we use private input vectors which can contain edge cases that were not covered by the test we gave you. These will not be made public prior to evaluation. For this module we require that you submit completed versions of "`tls_crypto`" (implemented in last weeks lab), "`tls_handshake.py`", "`tls_psk_state_machines`", and "`tls_psk_handshake`". You may modify any of the other files in the folder as you please. However, as a result of modifying files you may no longer get accurate evaluations of your own files, so we don't recommend this.

Summary of Evaluation Criteria. To summarize, you will be evaluated based on the correctness of your implementation of the individual functions:

1. `tls_13_client_hello` (4 points)
2. `tls_13_process_server_hello` (4 points)
3. `tls_process_server_cert_verify` (4 points)
4. `tls_process_finished` (4 points)
5. `tls_13_server_new_session_ticket` (4 points)
6. `tls_13_client_parse_new_session_ticket` (4 points)
7. `tls_13_client_prep_psk_mode_extension` (4 points)
8. `tls_13_client_add_psk_extension` (4 points)
9. `tls_13_server_parse_psk_extension` (4 points)

To evaluate the complete session resumption and 0-RTT we will run the server and client based on your submitted files. We will run this with and without trying to send early data in the following three configuratuion.

1. Your client implementation communicating with your server implementation. (6 points for PSK + 6 points for 0-RTT)
2. Your client implementation communicating with our server implementation. (3 points for PSK + 3 points for 0-RTT)
3. Our client implementation communicating with your server implementation. (3 points for PSK + 3 points for 0-RTT)

So a total of 60 points is available for this portion of this week's lab.

Submission Format

Your completed submission for week 5 should consist of *four* Python files, and should be named "`tls_crypto.py`" (which was part of last week's lab), "`tls_handshake.py`", "`tls_psk_state_machines.py`", and "`tls_psk_handshake.py`" respectively.

You are expected to upload your submission to Moodle. The submission for weeks 4 and 5 should be bundled into a single archive file named "module_2_submission_[insert LegiNo].zip". In summary, this file should contain the following files:

- "`sigma.py`", your implementation of the SIGMA protocol from the week 4 lab part I.
- "`tls_crypto.py`", your implementation of the cryptographic functions used by TLS 1.3 from the week 4 lab part II.
- "`tls_handshake.py`", your implementation of the TLS 1.3 Handshake from the week 5 lab.
- "`tls_psk_handshake.py`", your implementation of the PSK functionality of TLS 1.3 from the week 5 lab.
- "`tls_psk_state_machines.py`", your implementation of the TLS 1.3 state machine with PSK and 0-RTT support from the week 5 lab.

In conclusion, *happy coding*!

References

1. Eric Rescorla (2018) "RFC8446: The Transport Layer Security (TLS) Protocol Version 1.3" `https://tools.ietf.org/html/rfc8446`.

2. PyCryptodome. "Crypto.Hash Package" `https://pycryptodome.readthedocs.io/en/latest/src/hash/hash.html`

3. PyCryptodome. "Digital Signature Algorithm (DSA and ECDSA)" `https://pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html`

4. PyCryptodome. "PKCS#1 v1.5 (RSA)" `https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html`

5. PyCryptodome. "Modern modes of operation for symmetric block ciphers" `https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html`

6. PyCryptodome. "ChaCha20 and XChaCha20"

https://pycryptodome.readthedocs.io/en/latest/src/cipher/chacha20.html

7. Alex Moneger (2015) “tinyec 0.3.1”
https://pypi.org/project/tinyec/

Appendix: Additional Listings

Listing 16: TinyEC and Elliptic Curve Cryptography

```
from tinyec import registry
from Crypto.Cipher import AES
from Crypto.Protocol.KDF import HKDF
from Crypto.Hash import HMAC, SHA256, SHA512
from Crypto.Random import get_random_bytes
import math
import secrets

def compress(pubKey):
  return hex(pubKey.x) + hex(pubKey.y % 2)[2:]

def ec_setup(curve_name):
  curve = registry.get_curve(curve_name)
  return curve

def ec_key_gen(curve):
  sec_key = secrets.randbelow(curve.field.n)
  pub_key = sec_key * curve.g
  return (sec_key, pub_key)

def ec_dh(sec_key, pub_key):
  shared_key = sec_key * pub_key
  return shared_key
```

As you can see, a lot of the details of the underlying elliptic curve operations are hidden at this level. But you can still glean a high-level understanding of how elliptic-curve Diffie-Hellman key-exchange works, and compare it with the traditional variant of Diffie-Hellman key-exchange.

For key generation, an integer d is randomly sampled from $\mathbb{Z}_n$, where n (seen in Listing 1 as curve.field.n) is order of the point g (the point g generates the group of points on the elliptic curve). The integer d serves as the secret key `sk` in `ec_key_gen`. Then, the point g is added to itself d times to compute the public key `pk` in `ec_key_gen`. Computing an ECDH shared secret is again scalar multiplication and can be interpreted simply as adding the public-key `pk` to itself `sk`-many times.